ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»  
(СИБГУТИ)

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

“Структуры и алгоритмы обработки данных ”

на тему

«ДЕРЕВО КВАДРАНТОВ (Quadtree)»

|  |  |
| --- | --- |
| Выполнил студент | Гердележов Даниил Дмитриевич |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИВ-122 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  |  |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Оценка |  |  |  |
|  |  |  |  |

**Содержание**

[ВВЕДЕНИЕ 3](#_Toc121697776)

[1 Описание 4](#_Toc121697777)

[1.1 Определение 4](#_Toc121697778)

[1.2 Классификация 5](#_Toc121697779)

[1.3 Структура узла. 6](#_Toc121697780)

[1.4 Варианты использования. 6](#_Toc121697781)

[1.5 Анализ эффективности алгоритма. 8](#_Toc121697782)

[1.6 Основные шаги в операциях для дерева квадрантов. 9](#_Toc121697783)

[2 Экспериментальное исследование эффективности алгоритма. 10](#_Toc121697784)

[2.1 Организация исследования. 10](#_Toc121697785)

[2.2 Результаты исследования. 10](#_Toc121697786)

[ЗАКЛЮЧЕНИЕ 12](#_Toc121697787)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 13](#_Toc121697788)

[ПРИЛОЖЕНИЕ 15](#_Toc121697789)

[main.c 15](#_Toc121697790)

[point.h 17](#_Toc121697791)

[point.c 18](#_Toc121697792)

[bounds.h 18](#_Toc121697793)

[bounds.c 19](#_Toc121697794)

[node.h 20](#_Toc121697795)

[node.c 20](#_Toc121697796)

[quadtree.h 22](#_Toc121697797)

[quadtree.c 22](#_Toc121697798)

## ВВЕДЕНИЕ

Структуры данных «деревья» отлично подходят для хранения различных данных, так как являются ассоциативным массивом, потому что узлы хранят пары ключ-значение, с доступом к каждому за короткое время. Обычно скорость не хуже O(log(n)). Это добивается за счет того, что каждый узел дерева, помимо данных, хранит указатели на дочерние узлы (от двух и более, в зависимости от назначения) и оно строится по правилам. Базовое правило простейшего дерева: дочерний узел с ключом меньше (больше), чем у предка, становится его левым, а узел с большим (меньшим) – правым, потомком. В графическом виде простейшие деревья представляют ориентированный граф.

Общие операции:

* Вставка нового элемента
* Вставка поддерева
* Нахождение значения по ключу
* Перебор всех элементов дерева
* Удаление элемента
* Удаление поддерева

Существует множество модификаций «деревьев», в данной работе будет рассмотрена одна из них - «дерево квадрантов». По большей части ее применяют в алгоритме разбиения пространства. Так как каждый узел содержит четыре потомка, можно разбивать двумерное пространство ровно на четыре части, что по времени не очень затратно.

## 1 Описание

### 1.1 Определение

Дерево квадрантов (англ. quad tree), также называемое квадродеревом, – это дерево, в котором у каждого внутреннего узла ровно 4 потомка. Деревья квадрантов часто используются для рекурсивного разбиения двухмерного пространства по 4 квадранта (области), в таком случае области представляют собой квадратные (прямоугольные) участки двумерного пространства. Так же применяется для нахождения точек рядом с одиночной точкой путем поиска внутри области, окружающей данную точку.

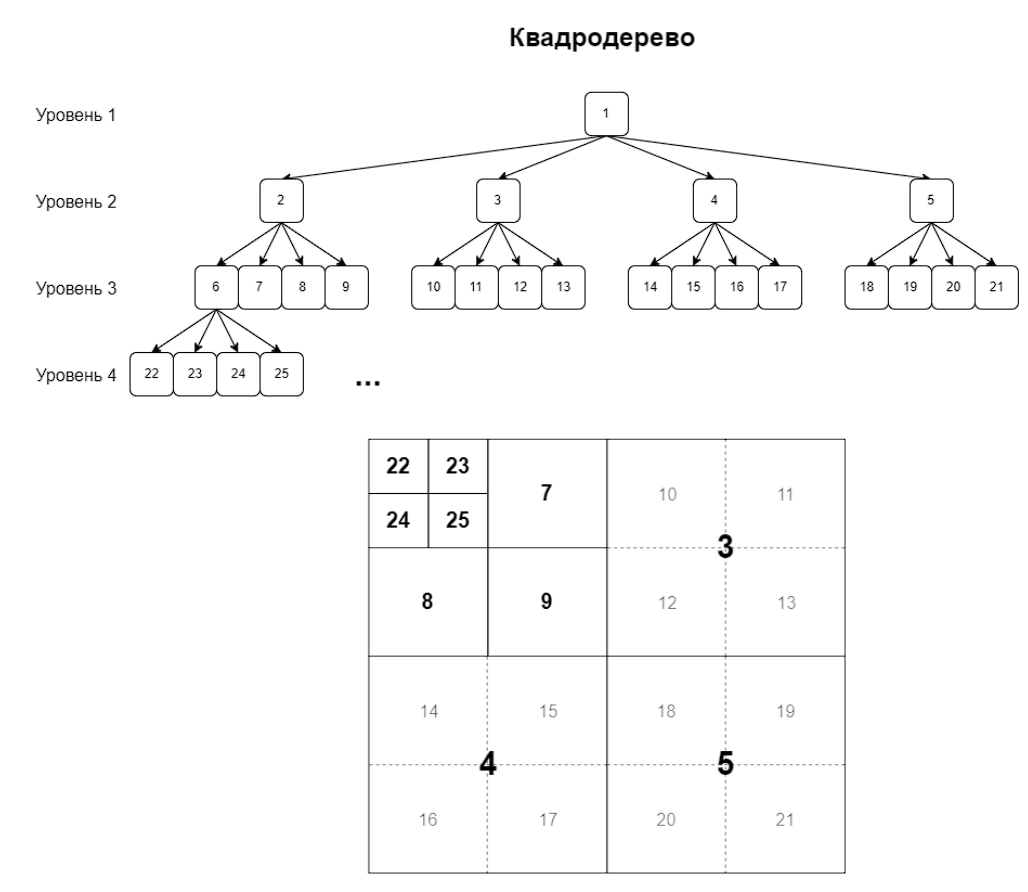


Рисунок 1 – Дерево квадрантов и разбитая с его помощью плоскость

Используя дерево квадрантов, можно эффективно выполнять поиск точек в двухмерном диапазоне, где эти точки определены координатами широты и долготы или декартовыми координатами (x, y). Дерево квадрантов хранит наборы координат в узлах и индексирует их по областям (ограничивающим прямоугольникам). Для поиска заданной пары координат нужно просматривать узлы дерева квадрантов.

### 1.2 Классификация

Деревья квадрантов могут быть классифицированы в соответствии с типом данных, который они представляют — пространством, точками, прямыми, кривыми. Также их можно разделить по тому, зависит ли форма дерева от порядка обработки данных. Некоторые виды деревьев квадрантов:

* + - 1. Region quadtree - Деревья квадрантов, разбивающие пространство, представляют какую-либо часть двумерного пространства разбивая его на 4 квадранта, субквадранты и так далее, причём каждый лист дерева соответствует определённой области. У каждого узла дерева либо 4 потомка, либо их нет вовсе (у листьев).

1. Point quadtree - Деревья квадрантов, хранящие информацию о точках, — разновидность бинарных деревьев, используемых для хранения информации о точках на плоскости. Форма дерева зависит от порядка задания данных. Использование таких деревьев очень эффективно в сравнении упорядоченных точек плоскости, причём время обработки равно O(log n).
2. Edge quadtree - Деревья квадрантов, хранящие информацию о линиях, используются для описания прямых и кривых. Кривые описываются точными приближениями путём разбиения пространства на очень мелкие ячейки.
3. Polygonal map quadtree - Деревья квадрантов, хранящие информацию о многоугольниках, могут содержать информацию о полигонах, в том числе и о вырожденных (имеющих изолированные вершины или грани)

Общие черты разных видов деревьев квадрантов:

* разбиение пространства на адаптирующиеся ячейки ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *adaptable cells*),
* максимально возможный объём каждой ячейки,
* соответствие направления дерева пространственному разбиению.

### 1.3 Структура узла.

Узел дерева квадрантов, хранящего информацию о точках плоскости, аналогичен узлу бинарного дерева лишь с той оговоркой, что узел первого имеет четыре потомка (по одному на каждый квадрант) вместо двух («правого» и «левого»). Ключ узла состоит из двух компонент (для координат *x* и *y*). Таким образом, узел дерева содержит следующую информацию:

* 4 указателя: NW (Северо-Запад), NE (Северо-Восток), SW (Юго-Запад), SE (Юго-Восток).
* Точка (center), описывающая центр квадранта.
* Границы (bounds) квадранта.
* Ключ (key) – содержимое узла.

### 1.4 Варианты использования.

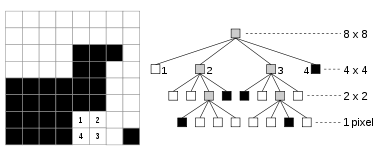
* Представление изображений.

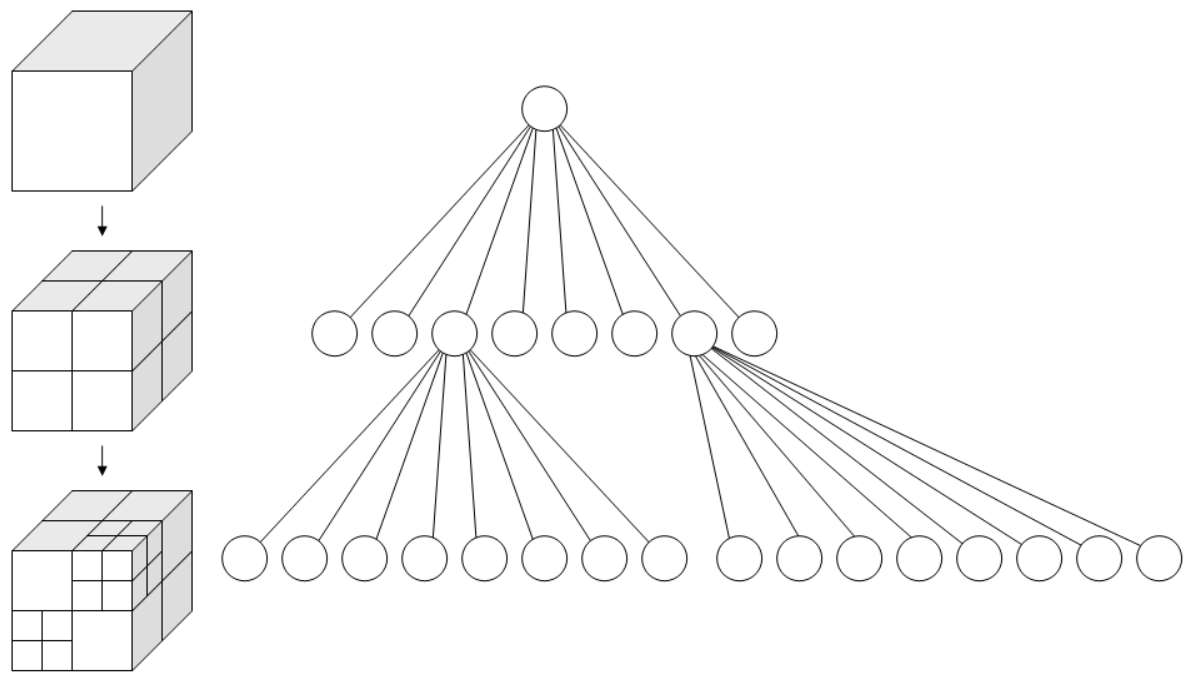
Рисунок 2 – Представление изображения с помощью дерева квадрантов

* Пространственные базы данных.
* Кодирование данных изображения.
* Эффективное обнаружение столкновений в двух измерениях.
* Отсечение невидимых частей ландшафта (*view frustum culling*).
* Хранение данных для табличных или матричных вычислений.
* Вычисления, связанные с многомерными полями (в вычислительной гидродинамике, электромагнетизме).
* Симуляция игры Жизнь.
* Вычисление состояний наблюдаемой динамической системы.
* Анализ частей фрактальных изображений (рис. 3).



Рисунок 3 - процесс восстановления изображения сжатого с помощью способа с применением дерева квадрантов.

Аналогом дерева квадрантов для разбиения трёхмерных пространств является октодерево (octree) (рис. 4).

Рисунок 4 - Слева: Рекурсивное разделение куба на октанты. Справа: Соответствующее октодерево.

### 1.5 Анализ эффективности алгоритма.

Основные операции для дерева квадрантов – это добавление и поиск точки.

Так же, как и для бинарного дерева поиска сложность в худшем случае составляет O(n), так как дерево может выродиться в связный список из-за того, что оно никак не балансируется. Однако при равномерно распределенных точках на плоскости достигается скорость O(logn).

Таблица 1 - Основные операции для дерева квадрантов и их сложность

|  |  |  |
| --- | --- | --- |
|  | В среднем | В худшем |
| Расход памяти | O(n) | O(n) |
| Поиск | O(log*n*) | O(n) |
| Вставка | O(log*n*) | O(n) |
| Удаление | O(log*n*) | O(n) |

Эффективность на примерах будет показана ниже.

### 1.6 Основные шаги в операциях для дерева квадрантов.

* Вставка
  + Найти квадрант, удовлетворяющий условию нахождения точки внутри границ и отсутствию иной точки в квадрате.
  + Если найдется такой квадрант, то вставить точку.
  + Иначе разбить этот квадрант на подквадранты.
* Поиск
  + Искать квадрант, пока не выполнится условие – искомая точка не выходит за границы, и она равна точке, содержащейся в квадранте.

## 2 Экспериментальное исследование эффективности алгоритма.

### 2.1 Организация исследования.

Измерения проводились на компьютере со следующими характеристиками: Ryzen 5 – 4500u (2.38GHz), 16Gb ROM 2667MHz, SSD M.2.

В дерево квадрантов последовательно добавлялось по 500 000 точек и производились замеры. Заполнение происходило равномерно.

### 2.2 Результаты исследования.

Таблица 2 – Результаты исследования функции добавления элемента.

|  |  |
| --- | --- |
| Количество точек, шт. | Среднее время, мкс |
| 500000 | 0.8725 |
| 1000000 | 0.4313 |
| 1500000 | 0.2916 |
| 2000000 | 0.2162 |
| 2500000 | 0.1705 |
| 3000000 | 0.1437 |
| 3500000 | 0.1206 |
| 4000000 | 0.1058 |

Таблица 3 – Результаты исследования функции поиска элемента.

|  |  |
| --- | --- |
| Количество точек, шт. | Среднее время, нс |
| 500000 | 0.0019073 |
| 1000000 | 0.0000840 |
| 1500000 | 0.0019073 |
| 2000000 | 0.0023841 |
| 2500000 | 0.0019073 |
| 3000000 | 0.0004010 |
| 3500000 | 0.0019073 |
| 4000000 | 0.0019073 |

Рисунок – 5 Зависимость времени выполнения операции добавления элемента от количества добавленных точек

Рисунок – 6 Зависимость времени выполнения операции поиска элемента от количества добавленных точек

## ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработана и исследована структура данных «Дерево квадрантов».

Осуществлено моделирование разработанной структуры данных.

Проведено исследование зависимости времени выполнения функций добавления и удаления элементов от количества добавленных элементов. Результаты исследования показали, что при увеличении количества точек операция добавления элемента выполняется за меньшее время, а время, затрачиваемое на выполнение операции поиска, увеличивается незначительно.

Показано, что скорость основных функций (добавления и поиска точки) совершается за логарифмическое время (**Θ**(log(n), где n – высота дерева)). Поэтому данная структура данных идеально подходит для таких профильных задач, как нахождение столкновений или пересечений объектов в двух измерениях, хранение данных для различных табличных или матричных вычислений, отсечение невидимых частей ландшафта и тд.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Wikipedia «Дерево квадрантов»* [электронный ресурс] // <https://is.gd/b2rQjR>
2. *Сычев С. А.* Применение дерева квадрантов в визуализации ландшафтов с изменяемым уровнем детализации / С. А. Сычев // Цифровое будущее инновационной экономики России : межвузовский сборник научных трудов и результатов совместных научно-исследовательских проектов. – Москва : Издательство "Аудитор", 2018. – С. 309-317. – EDN XMTRFJ.
3. Обработка информации и математическое моделирование : Материалы Российской научно-технической конференции, Новосибирск, 26–27 апреля 2018 года. – Новосибирск: Сибирский государственный университет телекоммуникаций и информатики, 2018. – 319 с. – ISBN 978-5-91434-042-8. – EDN XZGBUL.
4. *Educative «What is a Quadtree & how is it used in location-based services?»* [электронный ресурс] // <https://www.educative.io/answers/what-is-a-quadtree-how-is-it-used-in-location-based-services>
5. *«An interactive explanation of quadtrees.»* [электронный ресурс] // <https://jimkang.com/quadtreevis/>
6. *М. Г. Курносов, Д. М. Берлизов* АЛГОРИТМЫ И СТРУКТУРЫ ОБРАБОТКИ ИНФОРМАЦИИ // Новосибирск: Параллель, 2019 – 227 с.
7. *HABR «Деревья квадрантов и распознавание коллизий»* [электронный ресурс] // <https://habr.com/ru/post/473066/>
8. Симоненков, П. С. Способ фрактального сжатия изображений с модифицированной схемой покрытия ранговых блоков данными / П. С. Симоненков, К. И. Свириденков // Международный журнал информационных технологий и энергоэффективности. – 2017. – Т. 2. – № 3(5). – С. 45-50. – EDN ZIPMGD.
9. Воронцов К.В. Алгоритмы кластеризации и многомерного шкалирования. Курс лекций. МГУ, 2007. [Электронный ресурс]. – Режим доступа: http://www.ccas.ru/voron/download/Clustering.pdf, 01.05.2017
10. Иванов, В. Б. Пространственная кластеризация мест возникновения чрезвычайных ситуаций / В. Б. Иванов, В. Н. Гиляров, А. А. Мусаев // Труды СПИИРАН. – 2013. – № 1(24). – С. 108-115. – EDN PVKLJX.

## ПРИЛОЖЕНИЕ

### main.c

#include <stdio.h>

#include "quadtree.h"

#include <time.h>

#include <sys/time.h>

#include <stdlib.h>

#include <inttypes.h>

double wtime()

{

    struct timeval t;

    gettimeofday(&t, NULL);

    return (double)t.tv\_sec + (double)t.tv\_usec \* 1E-6;

}

int getrand(int min, int max)

{

    return (double)rand() / (RAND\_MAX + 1.0) \* (max - min) + min;

}

int main(void)

{

    srand(time(0));

    double NWx = 0.0;

    double NWy = 1000.0;

    double SEx = 1000.0;

    double SEy = 0.0;

    unsigned int sum\_point = (NWy / 0.5) \* (SEx / 0.5);

    Quadtree \*tree = quadtree\_new(NWx, NWy, SEx, SEy);

    double find\_500[8] = { 0 };

    double \*t = malloc(sizeof(double) \* sum\_point);

    double time\_500[8] = { 0 };

    int count\_find = 0;

    int count = 0;

    for (double i = 0; i < SEx; i += 0.5) {

        for (double j = 0; j < NWy; j += 0.5) {

            t[count] = wtime();

            quadtree\_insert(tree, j, i, i + j);

            t[count] = wtime() - t[count];

            count++;

            if (count % 500000 == 0) {

                for (int i = 0; i < 10000; i++) {

                    find\_500[count\_find] = wtime();

                    quadtree\_search(tree, getrand(0, 1000), getrand(0, 1000));

                    find\_500[count\_find] = wtime() - find\_500[count\_find];

                }

                printf("%.40f\n", find\_500[count\_find] / 500000);

                find\_500[count\_find] /= 500000;

            }

        }

    }

    unsigned int k = 0;

    for (unsigned int i = 0; i < 8; i++) {

        for (unsigned int j = k; j < k + 500000; j++) {

            time\_500[i] += t[j];

        }

        k += 500000;

    }

    double f[500] = { 0 };

    double j = 0.5;

    for (int i = 0; i < 500; i++) {

        f[i] = wtime();

        quadtree\_search(tree, j, i);

        f[i] = wtime() - f[i];

        j += 0.5;

    }

    FILE \*out = fopen("benchmark.txt", "w");

    double max = 0.0;

    double min = 99999.0;

    fprintf(out, "\nИзмерение времени добавления точек с шагом в 1:\n\n");

    for (int i = 0; i < sum\_point; i++) {

        if (t[i] > max) {

            max = t[i];

        }

        if (t[i] < min) {

            min = t[i];

        }

        fprintf(out, "%d.\t\t%0.10f\n", i, t[i]);

    }

    fprintf(out, "max = %.10f\n", max);

    fprintf(out, "min = %.20f\n\n\n", min);

    fprintf(out, "\nИзмерение времени поиска точек:\n\n");

    for (int i = 0; i < 500; i++) {

        fprintf(out, "%d.\t %.10f\n", i, f[i]);

    }

    fprintf(out, "\nСумма времени для добавления точек с шагом в 500 000:\n\n");

    int tmp = 500000;

    for (int i = 0; i < 8; i++) {

        fprintf(out, "%u-%u = %.10f\n", tmp - 500000, tmp, time\_500[i]);

        tmp += 500000;

    }

    fprintf(out, "\nСреднее время добавления точки с шагом в 500 000:\n\n");

    tmp = 500000;

    for (int i = 0; i < 8; i++) {

        if (i == 0) {

            fprintf(out, "%u-%u =\t\t %.10f\n", tmp - 500000, tmp, time\_500[i] / tmp);

        } else {

            fprintf(out, "%u-%u =\t %.10f\n", tmp - 500000, tmp, time\_500[i] / tmp);

        }

        tmp += 500000;

    }

    fprintf(out, "\nСреднее время поиска 1000 точек с шагом в 500 000:\n");

    for (int i = 0; i < 8; i++) {

        fprintf(out, "%d = %0.20f\n", i, find\_500[i]);

    }

    fclose(out);

    printf("capacity = %d\n", tree->capacity);

    quadtree\_free(tree);

    printf("\x1b[33m OK \x1b[0m \n");

    free(t);

    return 0;

}

### point.h

#ifndef POINT\_H

#define POINT\_H

typedef struct {

    double x;

    double y;

} Point;

Point \*point\_init(double /\*x\*/, double /\*y\*/);

void point\_free(Point \*/\*point\*/);

#endif

### point.c

#include <stdlib.h>

#include <assert.h>

#include "point.h"

Point \*point\_init(double x, double y)

{

    Point \*p = malloc(sizeof(Point));

    assert(p);

    p->x = x;

    p->y = y;

    return p;

}

void point\_free(Point \*p)

{

    free(p);

}

### bounds.h

#ifndef BOUNDS\_H

#define BOUNDS\_H

#include "point.h"

typedef struct {

    Point \*nw;

    Point \*se;

} Bounds;

Bounds \*bounds\_init(Point \*/\*nw\*/, Point \*/\*se\*/);

void bounds\_change(Bounds \*/\*bounds\*/, Point \*/\*nw\*/, Point \*/\*se\*/);

void bounds\_expand(Bounds \*/\*bounds\*/, double /\*x\*/, double /\*y\*/);

void bounds\_free(Bounds \*/\*bounds\*/);

#endif

### bounds.c

#include "bounds.h"

#include <stdlib.h>

#include <assert.h>

#include <stdio.h>

Bounds \*bounds\_init(Point \*NW, Point \*SE)

{

    assert(NW);

    assert(SE);

    Bounds \*bounds = malloc(sizeof(bounds));

    assert(bounds);

    bounds->nw = malloc(sizeof(Point));

    bounds->nw->x = NW->x;

    bounds->nw->y = NW->y;

    bounds->se = malloc(sizeof(Point));

    bounds->se->x = SE->x;

    bounds->se->y = SE->y;

    return bounds;

}

void bounds\_change(Bounds \*bounds, Point \*NW, Point \*SE)

{

    assert(bounds);

    assert(NW);

    assert(SE);

    if ((NW->x > SE->x) && (NW->y < SE->y)) {

        Point \*tmp = NW;

        NW = SE;

        SE = tmp;

    }

    bounds->nw->x = NW->x;

    bounds->nw->y = NW->y;

    bounds->se->x = SE->x;

    bounds->se->y = SE->y;

}

void bounds\_expand(Bounds \*bounds, double x, double y)

{

    assert(bounds);

    bounds->nw->x = (x < bounds->nw->x) ? x : bounds->nw->x;

    bounds->nw->y = (y > bounds->nw->y) ? y : bounds->nw->y;

    bounds->se->x = (x > bounds->se->x) ? x : bounds->se->x;

    bounds->se->y = (y < bounds->se->y) ? y : bounds->se->y;

}

void bounds\_free(Bounds \*bounds)

{

    if (!bounds) {

        return;

    }

    free(bounds->nw);

    free(bounds->se);

    free(bounds);

}

### node.h

#ifndef NODE\_H

#define NODE\_H

#include "bounds.h"

typedef struct Node {

    struct Node \*nw;

    struct Node \*ne;

    struct Node \*sw;

    struct Node \*se;

    Bounds \*bounds;

    Point \*center;

    int key;

} Node;

Node \*node\_new(void);

Node \*node\_with\_bounds(double /\*NWx\*/, double /\*NWy\*/, double /\*SEx\*/, double /\*SEy\*/);

void node\_free(Node \*/\*node\*/);

int node\_contains(Node \*/\*node\*/, Point \*/\*point\*/);

int node\_is\_empty(Node \*/\*node\*/);

#endif

### node.c

#include "node.h"

#include <assert.h>

#include <stdlib.h>

#include <stdio.h>

Node \*node\_new(void)

{

    Node \*node = malloc(sizeof(Node));

    assert(node);

    node->nw = NULL;

    node->ne = NULL;

    node->sw = NULL;

    node->se = NULL;

    node->bounds = NULL;

    node->center = NULL;

    return node;

}

Node \*node\_with\_bounds(double NWx, double NWy, double SEx, double SEy)

{

    Node \*node = node\_new();

    Point \*nw = point\_init(NWx, NWy);

    Point \*se = point\_init(SEx, SEy);

    node->bounds = bounds\_init(nw, se);

    point\_free(nw);

    point\_free(se);

    // node->center = point\_init((SEx + NWx) / 2, (NWy + SEy) / 2);

    return node;

}

void node\_free(Node \*node)

{

    point\_free(node->center);

    bounds\_free(node->bounds);

    if (node->nw) {

        node\_free(node->nw);

    }

    if (node->ne) {

        node\_free(node->ne);

    }

    if (node->se) {

        node\_free(node->se);

    }

    if (node->sw) {

        node\_free(node->sw);

    }

    free(node);

}

### quadtree.h

#ifndef QUADTREE\_H

#define QUADTREE\_H

#include "node.h"

typedef struct {

    Node \*root;

    int capacity;

} Quadtree;

Quadtree \*quadtree\_new(double /\*NWx\*/, double /\*NWy\*/, double /\*SEx\*/, double /\*SEy\*/);

void quadtree\_walk(Node \*/\*root\*/);

int quadtree\_insert(Quadtree \*/\*root\*/, double /\*x\*/, double /\*y\*/, int /\*key\*/);

// Point \*quadtree\_search(Quadtree \*/\*root\*/, double /\*x\*/, double /\*y\*/);

int split\_node(Quadtree \*/\*tree\*/, Node \*/\*node\*/);

int \_insert(Quadtree \*/\*tree\*/, Node \*/\*node\*/, Point \*/\*point\*/, int /\*key\*/);

Node \*get\_quadrant(Node \*/\*node\*/, Point \*/\*point\*/);

int node\_is\_leaf(Node \*/\*node\*/);

void quadtree\_free(Quadtree \*/\*tree\*/);

int quadtree\_search(Quadtree \*/\*tree\*/, double /\*x\*/, double /\*y\*/);

#endif

### quadtree.c

#include "quadtree.h"

#include <assert.h>

#include <stdlib.h>

#include <stdio.h>

Quadtree \*quadtree\_new(double NWx, double NWy, double SEx, double SEy)

{

    Quadtree \*tree = malloc(sizeof(Quadtree));

    assert(tree);

    tree->root = node\_with\_bounds(NWx, NWy, SEx, SEy);

    assert(tree->root);

    tree->root->nw = NULL;

    tree->root->ne = NULL;

    tree->root->sw = NULL;

    tree->root->se = NULL;

    tree->root->center = NULL;

    tree->capacity = 0;

    return tree;

}

void quadtree\_walk(Node \*node)

{

    if (node == NULL) {

        return;

    }

    if (node->bounds != NULL) {

        printf("{ nw.x:%.2f, nw.y:%.2f, se.x:%.2f, se.y:%.2f",

         node->bounds->nw->x, node->bounds->nw->y, node->bounds->se->x, node->bounds->se->y);

    }

    if (node->center) {

        printf(", center:%.2f, %.2f }:", node->center->x, node->center->y);

    } else {

        printf(" }:");

    }

    if (node->nw != NULL) {

        quadtree\_walk(node->nw);

    }

    if (node->ne != NULL) {

        quadtree\_walk(node->ne);

    }

    if (node->sw != NULL) {

        quadtree\_walk(node->sw);

    }

    if (node->se != NULL) {

        quadtree\_walk(node->se);

    }

    printf("\n");

}

int node\_contains(Node \*node, Point \*point)

{

    return node != NULL

        && node->bounds != NULL

        && node->bounds->nw->x <= point->x

        && node->bounds->nw->y >= point->y

        && node->bounds->se->x >= point->x

        && node->bounds->se->y <= point->y;

}

int node\_is\_empty(Node \*node)

{

    return node->nw == NULL

        && node->ne == NULL

        && node->sw == NULL

        && node->se == NULL

        && !node\_is\_leaf(node);

}

int node\_is\_pointer(Node \*node)

{

    return node->nw != NULL

        && node->ne != NULL

        && node->sw != NULL

        && node->se != NULL

        && !node\_is\_leaf(node);

}

int node\_is\_leaf(Node \*node)

{

    if (node == NULL) {

        return 1;

    }

    return node->center != NULL;

}

Node \*get\_quadrant(Node \*node, Point \*point)

{

    if (node == NULL) {

        return NULL;

    }

    if (node\_contains(node->nw, point)) {

        return node->nw;

    }

    if (node\_contains(node->ne, point)) {

        return node->ne;

    }

    if (node\_contains(node->sw, point)) {

        return node->sw;

    }

    if (node\_contains(node->se, point)) {

        return node->se;

    }

    return NULL;

}

int split\_node(Quadtree \*tree, Node \*node)

{

    Node \*nw;

    Node \*ne;

    Node \*sw;

    Node \*se;

    Point \*point\_old;

    int key\_old;

    double x = node->bounds->nw->x;

    double y = node->bounds->nw->y;

    double hw = (node->bounds->se->x - node->bounds->nw->x) / 2;

    double hh = (node->bounds->nw->y - node->bounds->se->y) / 2;

    nw = node\_with\_bounds(x, y, x + hw, y - hh);

    if (!nw) {

        return 0;

    }

    ne = node\_with\_bounds(x + hw, y, x + hw \* 2, y - hh);

    if (!ne) {

        return 0;

    }

    sw = node\_with\_bounds(x, y - hh, x + hw, y - hh \* 2);

    if (!sw) {

        return 0;

    }

    se = node\_with\_bounds(x + hw, y - hh, x + hw \* 2, y - hh \* 2);

    if (!se) {

        return 0;

    }

    node->nw = nw;

    node->ne = ne;

    node->sw = sw;

    node->se = se;

    point\_old = node->center;

    key\_old = node->key;

    node->center = NULL;

    node->key = 0;

    return \_insert(tree, node, point\_old, key\_old);

}

int \_insert(Quadtree \*tree, Node \*node, Point \*point, int key)

{

    if (node\_is\_empty(node)) {

        node->center = point;

        node->key = key;

        return 1;

    } else if (node->center) {

        if (node->center->x == point->x && node->center->y == point->y) {

            node->center = point;

            node->key = key;

        } else {

            if (!split\_node(tree, node)) {

                return 0;

            }

            return \_insert(tree, node, point, key);

        }

    } else if (node\_is\_pointer(node)) {

        Node \*quadrant = get\_quadrant(node, point);

        return quadrant == NULL ? 0 : \_insert(tree, quadrant, point, key);

    }

    return 0;

}

int quadtree\_insert(Quadtree \*tree, double x, double y, int key)

{

    Point \*point = point\_init(x, y);

    if (!point) {

        return 0;

    }

    int insert\_status = 0;

    if (!node\_contains(tree->root, point)) {

        point\_free(point);

        return 0;

    }

    if (!(insert\_status = \_insert(tree, tree->root, point, key))) {

        point\_free(point);

        return 0;

    }

    if (insert\_status == 1) {

        tree->capacity++;

    }

    return insert\_status;

}

void quadtree\_free(Quadtree \*tree)

{

    node\_free(tree->root);

    free(tree);

}

int quadtree\_search(Quadtree \*tree, double x, double y)

{

    Point \*point = point\_init(x, y);

    assert(point);

    if (!node\_contains(tree->root, point)) {

        return 1;

    }

    Node \*node = tree->root;

    while (!node\_is\_leaf(node)) {

        node = get\_quadrant(node, point);

    }

    if (node == NULL) {

        return 0;

    }

    return node->key;

}